A Parallel Multiobjective Particle Swarm Optimization Algorithm Based on Decomposition for Multicore Processors

***Abstract*—Multiobjective particle swarm optimization based on decomposition (MOPSO/D) is an effective algorithm for multiobjective optimization problems (MOPs). This paper proposes an island model-based parallel version of MOPSO/D algorithm using both MPI and OpenMP, which is abbreviated as MPI-OpenMP-MOPSO/D. It decomposes the whole population into several subspecies and strategies proposed for the design of network topology, exchanging particle information, and selecting parent particle are integrated into the algorithm. Based on the hybrid of distributed and shared-memory programming models, the proposed algorithm can fully use the processing power of today's multicore Processors and even a cluster. The experimental results show that MPI-OpenMP-MOPSO/D can achieve speedups of 2x on a personal computer equipped with a dual-core four-thread CPU. In terms of the quality of solutions, it can perform similarly to the serial MOPSO/D but greatly outperform NSGA-II. An additional experiment is done on a cluster, and the results show the speedup is not obvious for small-scale MOPs and it is more suitable for solving highly complex problems.**

***Index Terms*—multiobjective particle swarm optimization (MOPSO); decomposition; parallel computing; island model; MPI; OpenMP.**

I. Introduction

A multiobjective optimization problem (MOP) involves several objective functions which require simultaneous optimization. A MOP with n decision variables and m objective functions can be described as follows:

$$\min F(\mathbf{x}) = \left(f_1(\mathbf{x}), f_2(\mathbf{x}), \cdots, f_n(\mathbf{x})\right) \qquad (1)$$
$$\mathbf{x} = \left(x_1, x_2, \cdots, x_n\right) \in X \subseteq \mathbb{R}\text{, and}$$
$$g_j(\mathbf{x}) \le 0,\ j = 1, \cdots, m$$

where X is the n-dimensional variable search space, $F(\mathbf{x})$ is an m-dimensional objective space, $f_i(\mathbf{x}): X \to \mathbb{R}, i = 1, \cdots, m$ is the i-th objective function defined over X and $g_j(\mathbf{x}), j = 1, \cdots, m$ are m constraints to the variable search space. Since several objective functions be optimized simultaneously in a MOP, some of them cannot always be comparable or even conflict with each other. It is impossible to find an optimal solution in X which minimizes all the objectives simultaneously. For MOPs, the goal is to find a set of non-dominated solutions which is named as Pareto optimal solutions [1]. The set involving all the Pareto optimal solutions is Pareto Set (PS) and the set of all the corresponding objective function values of all solutions in PS is Pareto Front (PF).

Evolutionary algorithms (EAs) are a class of stochastic optimization methods by mimicking natural evolution. Because of the population-based searching way, EAs can balance search for both diversity and convergence. The population search possible solutions synchronously and increase search efficiency. Multiobjective evolutionary algorithms (MOEAs) are some variants of EAs, which are dedicated to solve MOPs by finding a well-distributed approximate of the Pareto Set.

Particle Swarm Optimization (PSO), developed by Eberhart and Kennedy [2], is an effective population-based stochastic optimization technique to solve Single-objective Optimization Problems (SOPs) [wenxian]. PSO is derived by the movement of a flock of birds, where every particle in the population flies based on both its own and the whole population’s movement. PSO supersedes other heuristic algorithms such as Genetic Algorithm (GA) in the simplicity of concept, ease of implement and efficiency of computation [wenxian]. Thus, PSO and many variants of it are introduced to solve Optimization Problems and Multiobjective Particle Swarm Optimization (MOPSO), a multiobjective variant of PSO, is proposed to solve MOPs.

Multiobjective Evolutionary Algorithm Based on Decomposition (MOEA/D), proposed by Qingfu Zhang and Hui Li, is an effective algorithm which adopts the main idea of that an approximation of the PF can be decomposed into a number of scalar objective optimization subproblems [wenxian]. Before MOEA/D, the majority of the MOEAs treat a MOP as a whole problem. Because some of objects in a MOP are always in conflict with each other, there are two big challenges in MOEAs: one is how to select global best (gBest) and personal best (pBest) and the other one is how to balance the convergence and diversity of the swarm to get an accurate and well-distributed approximation of the PF [wenxian].

Multiobjective Particle Swarm Optimization Based on Decomposition (MOPSO/D), a classic variant of MOEA/D, is implemented by PSO. It decomposes a MOP into a number of scalar objective optimization subproblems. Each particle in the swarm commits to finding a single optimal solution by learning from itself and its neighbors which defined based on the distances between their aggregation coefficient vectors. In MOPSO/D, gBest is updated by the best solution found so far and no pBest can be used, and population can be well-distributed because of the decomposition of the MOP.

With the increase of the number of objects in a MOP, it will become fairly complicated to solve. Due to the complicity of the MOP and the nature of MOPSO/D that needs large number of particles to decompose the problem and optimize every subproblem, which introduces high computational costs. In this case, many researchers focus on improving the algorithms' efficient and reducing the computation time through parallelization of evolutionary algorithms.

Many works in PSO parallelism have been introduced recently. Since most personal computers are now equipped with a multicore CPU and all cores in CPU support multithreaded programming, many proposed parallel PSO algorithms are based on multicores CPU with distributed memory using MPI for communication (MPI-PSO) and several parallel PSO implementations are based on OpenMP with shared memory (OpenMP-PSO). There is no parallel PSO implementation using hybrid of MPI and OpenMP that fully uses the processing power of today's multicore CPUs. For this reason, we modify the original MOPSO/D and implement a parallel version of it using MPI and OpenMP (MPI-OpenMP-MOPSO/D) and compare the performances of the parallel and the serial MOPSO/D. Compared to the sequential MOPSO/D, the proposed MPI-OpenMP-MOPSO/D has the following features.

- The algorithm adopts an island model and the entire population is divided into several subspecies based on the needed number of their neighbors in the algorithm. Each subspecies is regarded as an island and the all particles in an island are neighbors to each other. The evolution of one subspecies is computed by one core and the updating of a particle of the subspecies is done by one thread of the core, which can fully use the CPU power. If there are a large number of subspecies in a population, a computer cluster which is easy to be extended can be used to accelerate computation based on MPI.
- Since the data communication cost in distributed memory programming using MPI is costly and computation is relatively cheap, how to reduce MPI communication is the primary problem. In MPI-OpenMP-MOPSO/D, we reduce the communication of neighboring subspecies and we only transfer the information of two particles at left and right edges of a subspecies to its left and right neighbors, respectively.
- In order to further reduce the MPI communication cost, the proposed algorithm introduces subspecies best solution (sBest) and uses it to lead particles to fly. And it updates gBest using all sBests instead of the whole population every n iterations rather than every iteration.

We run sequential MOPSO/D and MPI-OpenMP-MOPSO/D on a multicore PC and a cluster, separately. The results show MPI-OpenMP-MOPSO/D can achieve a speedup of $2\times$ while maintain the nearly same quality of the final solutions compared with the serial MOPSO/D. To summarize, a parallel MOPSO/D algorithm is proposed in this paper, where the communication between subspecies is based on master-slave paradigm and the evolution of all particles in a subspecies is based on peer-to-peer mode. The proposed algorithm, applies hybrid of programming with MPI and OpenMP to develop a high efficient parallel MOPSO/D to solve MOPs and fully uses the power of multicore processors. Since the availability of MOPSO/D to MOPs and the speedup of mixed programming of MPI and OpenMP, MPI-OpenMP-MOPSO/D can solve highly complex MOPs based on the computational power of computer clusters.

This paper is organized as follows. In Section II, a brief review on MOEA/D and its one implementation based on PSO – MOPSO/D are presented. Section III introduces MPI-OpenMP-MOPSO/D. Section IV and V compare MPI-OpenMP-MOPSO/D with sequential MOPSO/D and MPI-MOPSO/D and OpenMP-MOPSO/D, respectively. Section VI includes the discussion, conclusions, and future research.

II. MOPSO/D REVIEW

MOEA/D decomposes a MOP into N scalar optimization subproblems and solves them simultaneously. And there are several ways to decompose the problem. In this section, we first present a superior decomposing approach named Tchebycheff Approach briefly. In the following, we introduce MOPSO/D algorithm employing Tchebycheff decomposition approach.

Three major decomposition approaches to MOPs are Weighted Sum Approach, Tchebycheff Approach, and Boundary Intersection Approach [wenxian]. According to the experiments of [wenxian], MOPSO/D employing Tchebycheff decomposition approach obtains a best solution of MOPs comparing to the others. Since we are not studying the decomposition approaches, we only introduce Tchebycheff Approach employed in MOPSO/D and three parallel variants of it in this paper.

In this approach, let $\boldsymbol{\omega}^1,\cdots,\boldsymbol{\omega}^N$ is a set of uniform spread of N weight vectors of m objects, which are used to decompose the MOP into N scalar optimization subproblems. $\boldsymbol{\omega}^j=\left(\omega_1^j,\cdots,\omega_m^j\right)^T$ for each $j=1,\cdots,N$ where $\omega_i^j\geq 0$ for all $i=1,\cdots,m$ and $\sum_{i=1}^m\omega_i^j=1$. Then a scalar optimization subproblem is of the form

$$g(\mathbf{x}\,|\,\boldsymbol{\omega}^{\mathbf{j}},\mathbf{z}^{*})=\text{minimize}\max_{i=1,\cdots,m}\left[\omega_i^j\left|f_i(\mathbf{x})-z_i^*\right|\right]\qquad(2)$$
$$\text{subject to }\ \mathbf{x}\in X$$

where $z_i^*=\max\left\{f_i(\mathbf{x})\,|\,\mathbf{x}\in X\right\}$ and $\mathbf{z}^*=\left(z_1^*,\cdots,z_m^*\right)^T$. z_i^* is the best fitness value named reference point found by the population so far. Each $\mathbf{x}$ that satisfies $f_i(\mathbf{x})=z_i^*$ is a best solution for the $i-th$ objective function. The optimal solution $\mathbf{x}$ are written $gBest_i$ in MOPSO/D and a gBest with size of m is used to store all global best solutions found so far.

The sequential MOPSO/D works as follows:

Algorithm 1: MOPSO/D

```
Input: 1) MOP with $m$ objective functions;
       2) The n-dimensional search space $X$;
       3) The number of subproblems( the number of population) $N$;
       4) The uniform spread of $N$ weight vectors $\boldsymbol{\omega}^j, j=1,\cdots,N$;
       5) The number of neighbors of a weight vector $T$;
       6) A stopping criterion.
Output: All solutions of the population to $N$ subproblems.
  Step 1) Initialization
    1.1) Initialize the gBest with $\varnothing$;
    1.2) Select $T$ neighbors for each particle based on the Euclidean distances between their weight
         vectors and store them into $B(p)=\{p_1,\cdots,p_T\}, p=1,\cdots,N$;
    1.3) Randomly initialize $\mathbf{x}^1,\cdots,\mathbf{x}^N$ in the population;
    1.4) Compute the initial objective fitness values $F(\mathbf{x}^i)=[f_1(\mathbf{x}^i),\cdots,f_m(\mathbf{x}^i)]$;
    1.5) Generate the initial reference point vector $\mathbf{z}^*=\left(z_1^*,\cdots,z_m^*\right)^T$ by comparing the initial objective
         fitness values.
  Step 2) Update
    For $i=1\ to\ N$
      2.1) Randomly select two neighbors from $B(i)$, and generate a new solution $\mathbf{y}$ using a genetic
           algorithm based on them and $\mathbf{x}^i$;
      2.2) Produce a new solution $\mathbf{y}'$ by applying a heuristic improvement on $\mathbf{y}$;
      2.3) If the reference point $z_j^*<f_j(\mathbf{y}')$, for each $j=1,\cdots,m$, then update $z_j^*=f_j(\mathbf{y}')$ and
           $gBest_j=\mathbf{y}'$;
      2.4) For each neighbor $j(j=1,\cdots,T)$ of particle $i$, if $g(\mathbf{y}'|\boldsymbol{\omega}^j,\mathbf{z}^*)<g(\mathbf{x}^j|\boldsymbol{\omega}^j,\mathbf{z}^*)$, then update
           $\mathbf{x}^j=\mathbf{y}'$;
    End For
  Step 3) Check Stopping Criteria
    Stop the algorithm and output all optimal solutions if the stopping criteria is satisfied. Otherwise, go
```

to Step 2).

III. MPI-OPENMP-BASED MOPSO/D

In order to fully use the processing power of multicore CPU, we adopt MPI and OpenMP to accelerate the computation of MOPSO/D and propose a variant of it based on MPI and OpenMP (MPI-OpenMP-MOPSO/D). In this section, we analyze the steps of serial MOPSO/D algorithm which can be executed in parallel and appropriately modify it to make parallel computing easier.

A. Network Topology and Parallelism

A mix of master-slave paradigm and peer-to-peer model can apply to the proposed parallel algorithm on multi cores. Population are divided into S subspecies and every subspecies contains T $(T = N / S)$ internal particles to solve T scalar optimization subproblems and K external particles (EP) to store the exchanged particles from two neighbor subspecies. Each subspecies is regarded as an island and the computation of it can be done by a process. These T particles in a subspecies are neighbors to each other. In order to simplify the problem, we assume that N is divisible by S.

In the master-slave paradigm, Process 0 is the master node and each other process is a slave node. All global information of population should be stored and updated by Process 0 and all other steps can be synchronously executed should be done by each process in parallel. Thus, in initialization, gBest is maintained by Process 0 and it calculates all weight vectors for population and distributes the needed components to each of the other processes using MPI_Scatter function. The initialization of gBest and computation of weight vectors are executed sequential.

Since a process does evolution computation for a subspecies and a thread does it for a single particle in a subspecies, both Step 1.3 and 1.4 of Algorithm 1 can be parallel executed on thread level, while Step 1.5 can only be parallel executed on process level.

Theoretically, all updates of the population in Step 2 can be executed in parallel. However, since nearly every particle need exchange solution information with other particles in the different processes, the communication costs between particles are enormous. Thus, the topology of information transmission of particles can be revised to reduce communication costs on the basis of ensuring the optimization results.

In order to reduce communication costs between particles in neighbor processes, each process maintains K external particles (**EP**) copied from neighbors. Half of **EP** come from the left neighbor process and the others come from the right one. If a process has no left or right neighbor (For example, Process 0 has no left neighbor.), this process should copy its first or last $K / 2$ particles and store them into the corresponding positions in **EP**, respectively. In this sense, all processes are peers. The network topology of MPI-OpenMP-MOPSO/D is represented in Figure 1.

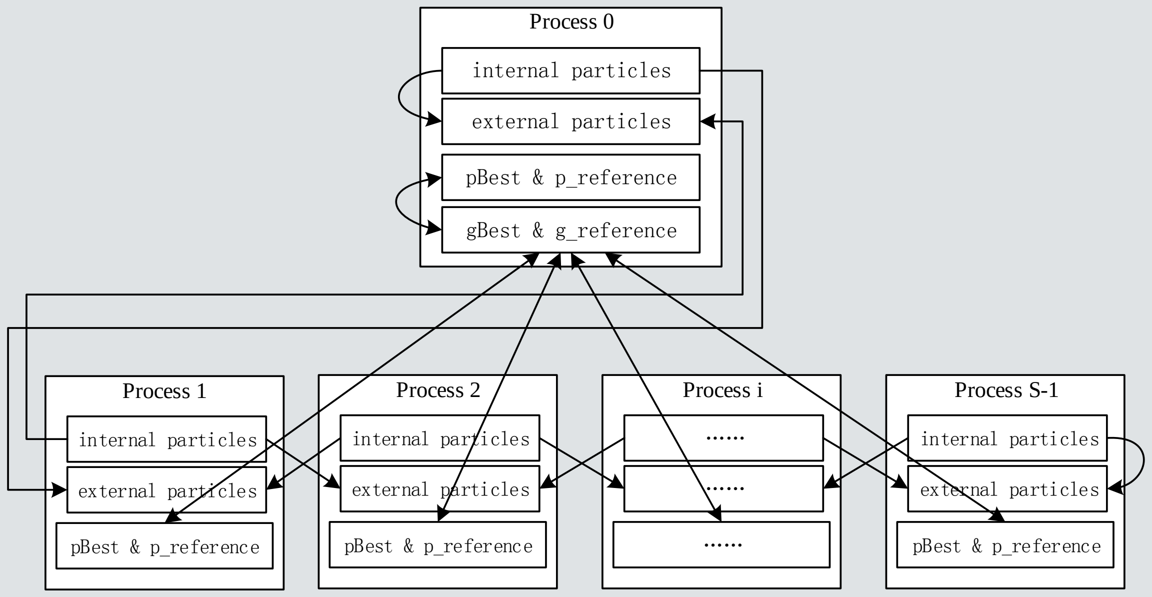


Fig. 1. Topology of MPI-OpenMP-MOPSO. The lines with arrows represent the information transmission. A mix paradigm of master-slave paradigm and peer-to-peer model is applied to the architecture of MPI-OpenMP-MOPSO

The proposed parallel algorithm uses not only two random neighbor particles but **EP** to generate a new solution $\mathbf{y}$. Then a perturbator operator is employed to help the particle to escape from local **PF** and obtain a newer solution $\mathbf{y}'$. Both of the two steps can be parallel executed for every particle on a thread level and the generation strategy is specifically introduced in Part C.

Like gBest, a subspecies best (sBest) should be maintained by each subspecies to lead the particles to fly. So the sBest and the corresponding subspecies reference point are updated after generating a new solution every time. The gBest and global reference point, computed by comparing all sBests, can be updated every several iterations to reduce communication costs. If a neighbor's current solution is worse than the new solution of the particle, then update it with the new solution. The update of sBest and gBest can be executed on process level while updating neighbors can be done in thread level, which is more efficient.

At last step, the algorithm checks if the stopping criteria is satisfied. If it is satisfied, then stop it and output results; otherwise, go to Step 2.

The algorithm is presented in Algorithm 2 and the flow chart of MPI-OpenMP-MOPSO/D is shown in Figure 2.

Algorithm 2: MPI-OpenMP-MOPSO/D

```
Input: 1) MOP with $m$ objective functions;
       2) The n-dimensional search space $X$;
       3) The number of subproblems $N$;
       4) The uniform spread of $N$ weight vectors $\boldsymbol{\omega}^j, j = 1,\cdots,N$;
       5) The number of subspecies $S$;
       6) The number of particles in a subspecies $T$, which is equal to $N/S$;
       7) The number of particles to exchange between two neighbor subspecies $K$.
       8) A stopping criterion.
Output: All solutions of the population to $N$ subproblems.
  Step 1) Initialization
    1.1) Initialize the gBest of process 0 with $\varnothing$;
    1.2) Initialize the sBest of all processes with $\varnothing$ in parallel;
    1.3) Process 0 calculates all weight vectors for population and distributes the needed components to each of the other processes using MPI_Scatter function.
    1.4) All particles in a process are its neighbors for each particle and store them into $B(p) = \{p_1,\cdots,p_T\}, p = 1,\cdots,N$;
    1.5) Randomly initialize $\mathbf{x}^1,\cdots,\mathbf{x}^S$ for each subspecies in parallel;
    1.6) Compute the initial objective fitness values $F(\mathbf{x}^i) = [f_1(\mathbf{x}^i),\cdots,f_m(\mathbf{x}^i)]$ in parallel;
    1.7) Update sBest and the subspecies reference point vector $\mathbf{s}^* = (s_1^*,\cdots,s_m^*)^T$ by comparing the initial objective fitness values in parallel.
    1.8) Update gBest and the global reference point vector $\mathbf{z}^* = (z_1^*,\cdots,z_m^*)^T$ by comparing sBest.
  Step 2) Update
    For $i = 1$ to $N$
      2.1) Swap particles with left and right neighbor subspecies for each subspecies and store them into EP.
      2.2) Randomly select two neighbors from $B(i)$, and generate a new solution $\mathbf{y}$ by using evolution strategy in parallel;
      2.3) Produce a new solution $\mathbf{y}'$ by applying a perturbation on $\mathbf{y}$;
      2.4) If the subspecies reference point $s_j^* < f_j(\mathbf{y}')$, for each $j = 1,\cdots,m$, then update $s_j^* = f_j(\mathbf{y}')$ and $sBest_j = \mathbf{y}'$;
      2.5) Update gBest and the global reference points.
```

```
        2.6) For each neighbor $j(j=1,\cdots,T)$ of particle $i$, if $g\left(\mathbf{y}'|\boldsymbol{\omega}^j,\mathbf{z}^*\right)<g\left(\mathbf{x}^j|\boldsymbol{\omega}^j,\mathbf{z}^*\right)$, then update
        $\mathbf{x}^j=\mathbf{y}'$;
    End For
**Step 3) Check Stopping Criteria**
    Stop the algorithm and output all optimal solutions if the stopping criteria is satisfied. Otherwise, go to Step 2).
```

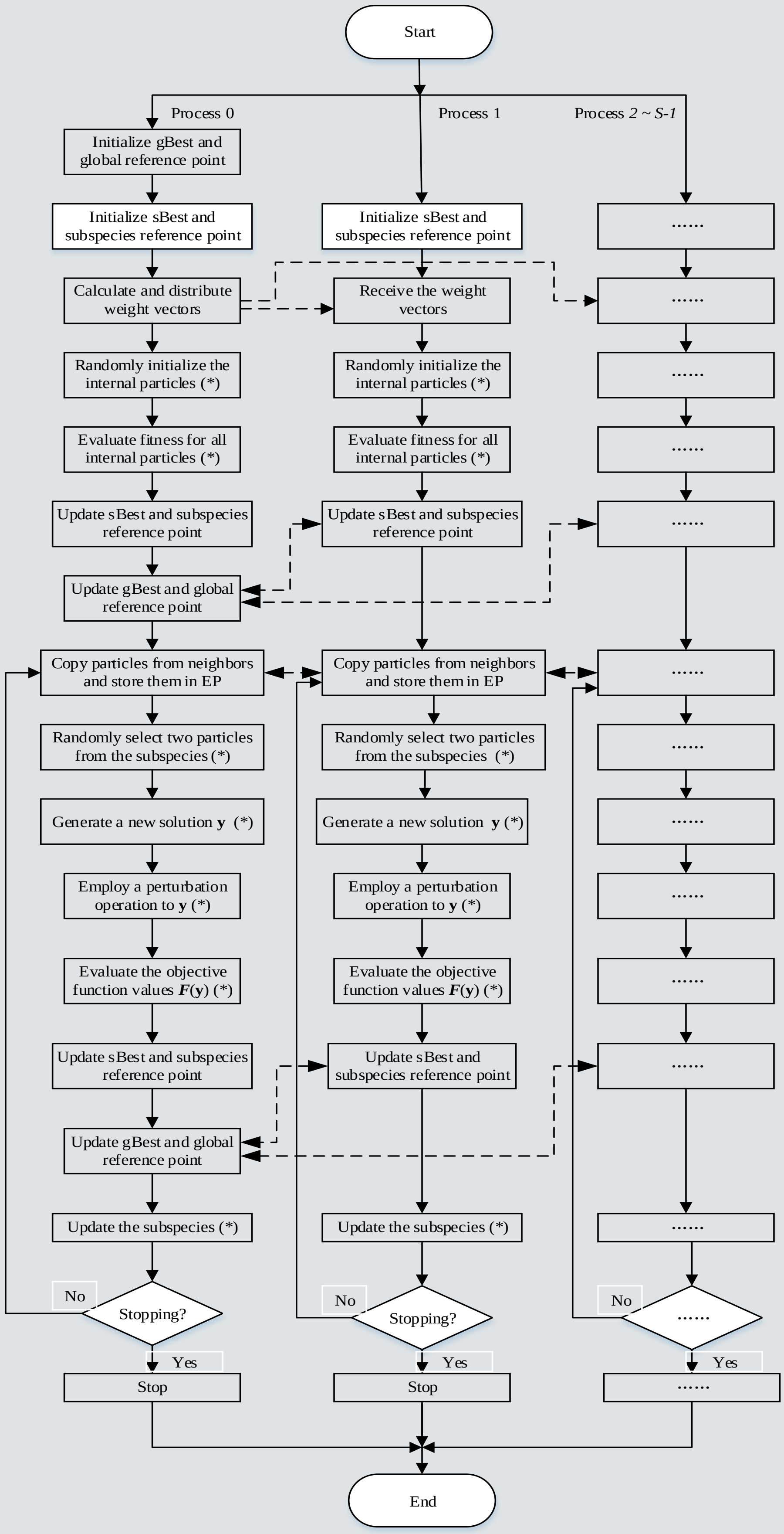

Start
Process 0
Process 1
Process 2 ~ S-1
Initialize gBest and global reference point
Initialize sBest and subspecies reference point
Calculate and distribute weight vectors
Receive the weight vectors
Randomly initialize the internal particles (*)
Evaluate fitness for all internal particles (*)
Update sBest and subspecies reference point
Update gBest and global reference point
Copy particles from neighbors and store them in EP
Randomly select two particles from the subspecies (*)
Generate a new solution y (*)
Employ a perturbation operation to y (*)
Evaluate the objective function values F(y) (*)
Update the subspecies (*)
Stopping?
No
Yes
Stop
......
End

Fig. 2. Flow chart of MPI-OpenMP-MOPSO/D. The solid lines represent the logical flow and the dashed arrows represent the information transmission. A step marked with a star means it can execute in parallel on thread-level.

B. Particles Exchange Strategy

As is shown in Figure 1, each subspecies exchanges its K particles with two neighbors. For one subspecies, its left half particles are close to its left neighbor subspecies while its right half particles are close to its right neighbor. Thus, when it exchanges particles with its neighbors, $K/2$ particles which are sent to its left neighbor should be randomly selected from its left half particles and the other $K/2$ particles that are sent to its right neighbor should be randomly selected from its right half particles.

After selecting the particles to exchange, there are two communication model to achieve the information transmission using MPI: one is point-to-point communication and the other one is collective communication. The point-to-point communication manner is intuitive that Process i send its information to the target processes, then the target processes receive the information. However, there are several disadvantages about point-to-point communication manner using MPI.

- An unreasonable order of sending and receiving information for processes can always cause system deadlock. But it is extremely difficult to design a reasonable flow of communication when there are many point-to-point communications between the processes.
- Compared with collective communications, point-to-point communications are inefficient, particularly when there exist extensive point-to-point communications.
- Programming using point-to-point communications is more complex, cumbersome and error-prone.

Based on above points, we adopt collective communications to achieve the particles exchange between neighbor subspecies in the proposed algorithm. In the master-slave communication paradigm employed in this paper, Process 0 is the master node and the other processes are slave nodes. Thus, the master node (Process 0) gathers the information used to exchange with neighbors from all slave nodes using MPI_Gather function. After briefly shifting the information, the master node distributes them to the corresponding processes.

C. Selecting Parent Particles

In the island model of MPI-OpenMP-MOPSO/D, all particles in an island are regarded as its neighbors for each other. However, the true nearness relationships among all particles in the population are defined by the distances between their weight vectors of their subproblems. And the optimal solution of two near subproblems should be similar. In order to avoid particles in an island losing subspecies diversity and falling into a local best point, the exchanged particles and two randomly selected neighbors are used to generate a new solution for every particle.

It is a more reasonable strategy in particle generation that selecting parent particles from neighbor particles or exchanged particles and even selecting which one from the whole exchanged particles should be based on the position of the particle in an island. At the first step, a probability is defined based on its position to decide where to select the parent particles used to generate a new solution.

$$\Pr = \begin{cases} \dfrac{T/2-p}{T-p} & if \;\; p < T/2 \\ 0 & if \;\; p = T/2 \\ 1-\dfrac{T}{2p} & if \;\; p > T/2 \end{cases} \qquad (3)$$

where $p = 0,\cdots,T-1$ is the position of the particle in its subspecies and T is the number of particles in a subspecies defined in part A. So, a parent particle is selected from EP and the other one is selected from two randomly selected neighbors if a random value is less than $\Pr$, whereas the parent particles are both selected from neighbors at the rate of $1-\Pr$.

At the second step, if one parent particle should be selected from EP, it is decided by the particle position that selecting the particle from the first or last half of EP. If $p < T/2$, the target region is the first half of EP, otherwise, that is the last half of it. Then randomly select a particle from the target region as a parent.

D. Discussions

1) How to set T and K: T is the size of particles in a subspecies and K is the number of exchanged particles between two neighbor subspecies. In the proposed algorithm, only the exchanged particles and neighbor particles are used to generate a new solution for a subproblem. So the setting of T and K is important for the optimal solutions. If T is too small, two neighbor particles selected in Step 2.2 may be too similar and the offspring particles will quickly converge to a local best point. Meanwhile, If T is too large, those two neighbor particles may be quite different from the current subproblem, so their offspring can hardly find the best solution of the subproblem. Since the exchanged particles coming from neighbor subspecies, it ensures the diversity of subspecies. However, if K is too large, much information need be transferred from one subspecies to another subspecies and the communication costs will be extensive. Actually, the EP with small K can lead to enough ability to explore the new solution areas.

2) Why the evolution of a subspecies is executed by a single process: In the MPI-OpenMP-MOPSO/D, a process creates T threads and a thread executes the evolution of a particle. Based on the CPU's power, several threads can run in a process simultaneously. However, the number of threads that a process allows to execute simultaneously is much less than the number of particles in a subspecies. So these threads are divided into several groups and they run in parallel within a group and run sequentially among the groups. In order to make all particle evolution can be executed in parallel, a subspecies can be divided into several parts and each part evolves on a separate process. The neighbors of a particle will lived in several islands and it will result in huge cost of communication when the particle needs the information of its neighbors living in a different island. Moreover, the separation of neighbors will bring enormous difficulties for programming.

IV. Comparision Of MPI-OpenMP-MOPSO/D With Serial MOPSO/D

In this section, five benchmark functions are applied to test the parallel MPI-OpenMP-MOPSO/D algorithm and sequential MOPSO/D. Based on the results of the experiment, we evaluate the performance of MPI-OpenMP-MOPSO/D and compare it with sequential MOPSO/D.

A. Multiobjective Test Instances

Bi-objective ZDT and 3-objective DTLZ test instances are widely used to test the performance of MOEAs in many researches [wenxian]. These test instances can effectively test if a MOEA is able to maintain a good diversity of the population and the particles can quickly converge to the PF or not. Since the experiment result can be significantly influenced by how to construct subspecies from population and this experiment focuses on the feasibility of parallelization of MOPSO/D, five ZDT test instances are used to compare their different performances between MPI-OpenMP-MOPSO/D and sequential MOPSO/D. Five ZDT test instances are as follows.

- ZDT1

$$\text{target function: } \begin{cases} \min\ f_1(x) = x_1 \\ \min\ f_2 = g(x)\left(1 - \sqrt{x_1 / g(x)}\right) \\ where\ g(x) = 1 + 9\sum_{i=2}^{n} x_i / (n-1) \end{cases}$$

Search Space: $x \in [0,1]^{30}$ and $n = 30$. Its PF is convex.

- ZDT2

$$\text{target function: } \begin{cases} \min\ f_1(x) = x_1 \\ \min\ f_2 = g(x)\left(1 - (x_1 / g(x))^2\right) \\ where\ g(x) = 1 + 9\sum_{i=2}^{n} x_i / (n-1) \end{cases}$$

Search Space: $x \in [0,1]^{30}$ and $n = 30$. Its PF is nonconvex.

- ZDT3

$$\text{target function: } \begin{cases} \min f_1(x) = x_1 \\ \min f_2 = g(x)\left(1 - \sqrt{x_1 / g(x)} - x_1 / g(x)\sin(10\pi x_1)\right) \\ \textit{where } g(x) = 1 + 9\sum_{i=2}^{n} x_i / (n-1) \end{cases}$$

Search Space: $x \in [0,1]^{30}$ and $n = 30$. Its PF is disconnected.

- ZDT4

$$\text{target function: } \begin{cases} \min f_1(x) = x_1 \\ \min f_2 = g(x)\left(1 - \sqrt{x_1 / g(x)}\right) \\ \textit{where } g(x) = 1 + 10(n-1) + \sum_{i=2}^{n}\left(x_i^2 - 10\cos(4\pi x_i)\right) \end{cases}$$

Search Space: $x \in [x_1, \cdots, x_{10}],\ x_1 \in [0,1],\ x_i \in [-5,5](i = 2, \cdots n)$ and $n = 10$. It has many local PFs.

- ZDT6

$$\text{target function: } \begin{cases} \min f_1(x) = 1 - \exp(-4x_1)\sin^6(6\pi x_1) \\ \min f_2 = g(x)\left(1 - (f_1(x) / g(x))^2\right) \\ \textit{where } g(x) = 1 + 9\left(\sum_{i=2}^{n} x_i / (n-1)\right)^2 \end{cases}$$

Search Space: $x \in [0,1]^{30}$ and $n = 30$. Its PF is nonconvex. With the closing to the PF, the density of the solutions becomes very small.

B. Experimental Platform

To fairly compare the different performances of sequential MOPSO/D and MPI-OpenMP-MOPSO/D, we run these two algorithms on a personal computer, separately. The details of computer configuration are as follows.

- An Intel Core dual-core 4-thread i3-4150 CPU at 3.5 GHz.
- 8 GB main memory.
- MS Windows 7 Professional operating system.

The above is the main hardware in this experimental personal computer. In terms of software, MPI and OpenMP are necessary for the experiment. MPI is a cross-language communication protocol and it defines some message passing interface to write parallel programs easily. So MPI has many different implementations. MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. [wenxian] and MPICH2 v.1.4.1p1 for Windows are used to implement MPI-OpenMP-MOPSO/D. The Parallel and serial programs are compiled by VS2010, respectively. The OpenMP version which shipped with VS2010 is used in the implementation of the proposed algorithm.

C. Performance Metrics

The speedup Sp is used to present the speed performance of a parallel algorithm and it is given by formula (4).

$$Sp = \frac{T_{serial}}{T_{parallel}} \qquad (4)$$

where T_{serial} and $T_{parallel}$ are the run-time of the serial algorithm and the parallel algorithm's run-time, respectively.

Inverted Generational Distance (IGD) is applied in this paper to quantify the performances of MOPSO/D and MPI-OpenMP-MOPSO/D. IGD measures the distance between the real Pareto front and the approximate

Pareto front obtained by an algorithm, and the distance is used to evaluate the convergence and diversity of the approximate PF. IGD can be calculated by formula (5),

$$IGD = \frac{1}{|P|}\sum_{i=1}^{|P|} Dist_i \qquad (5)$$

where $Dist_i = \min_{j=1}^{|A|}\sqrt{\sum_{m=1}^{M}\left(\frac{f_m(\mathbf{p}_i) - f_m(\mathbf{a}_j)}{f_m^{\max} - f_m^{\min}}\right)^2}$, $f_m^{\max}$ and $f_m^{\min}$ are the max and min values on the m-th object in set P, respectively. $m = 1,\cdots,M$, P is the real Pareto Solution Set and $\mathbf{p}_i, i = 1,\cdots,|P|$ is the $i-th$ solution in P. A is the approximate Pareto Solution Set and $\mathbf{a}_j, j = 1,\cdots,|A|$ is the $j-th$ solution in A.

D. Experimental Setting

The size of population N is set to be 100 for both algorithms for all test instances. Because the information communication among subspecies is few in the island model, the speed of convergence of MPI-OpenMP-MOPSO/D may decrease to a certain extent. In order to guarantee the quality of solutions, the number of particles generations needs to increase accordingly. Table I shows the number of generations for each test instance.

TABLE I. GENERATIONS OF TEST INSTANCES FOR TWO ALGORITHMS

Test Instance	Number of generations in	
	sequential MOPSO/D	MPI-OpenMP-MOPSO/D
ZDT1	300	500
ZDT2	300	500
ZDT3	300	500
ZDT4	500	1500
ZDT6	500	1500

Except the changes in the algorithm structure and the way to select parents for each particle, MPI-OpenMP-MOPSO/D does not change the other parts of the original MOPSO/D algorithm. In order to reduce the communication costs between processes, the number of particles to exchange with two neighbor subspecies K is set to be 2. The number of particles in a subspecies T is set to be 20 and the weight vectors $\boldsymbol{\omega}^j, j = 1,\cdots,N$ are set based on Section II and III. The size of the real Pareto Solution Set is set to be 1000 and the points in it are constructed uniformly.

In population initialization, particles are generated by uniformly stochastic sampling. After selecting two parent particles in both algorithms, their offspring are separately generated by a same simulated binary crossover. Then the offspring are modified by a same perturbation operator. According to [wenxian], the distribution indexes in crossover operator and the perturbation operator are both 20, the crossover rate is set to be 1.00 and mutation rate is set to be $1/D$, where D is the dimension of variables. For each test instance, the above two algorithms are executed 30 times independently.

E. Experimental Results

Table II shows the average wall clock time of MPI-OpenMP-MOPSO/D and the average CPU time of serial MOPSO/D and the average speedup for each test instance. In general, MPI-OpenMP-MOPSO/D can reduce the run-time for all test instances. MPI-OpenMP-MOPSO/D runs about twice as fast as serial algorithm for ZDT1, ZDT2 and ZDT3, and only about 1.2 times for ZDT4 and ZDT6. However, in consideration of the number of generations showed in Table I, MPI-OpenMP-MOPSO/D can obtain a 3.2~3.5 times computing speedup for each instance, which is consistent with the dual-core 4-thread CPU used in the experiment.

TABLE II. AVERAGE RUN-TIME AND SPEEDUP

Test Instance	Average run-time (in millisecond) used by		Speedup	Computing Speedup
	serial MOPSO/D	MPI-OpenMP-MOPSO/D		

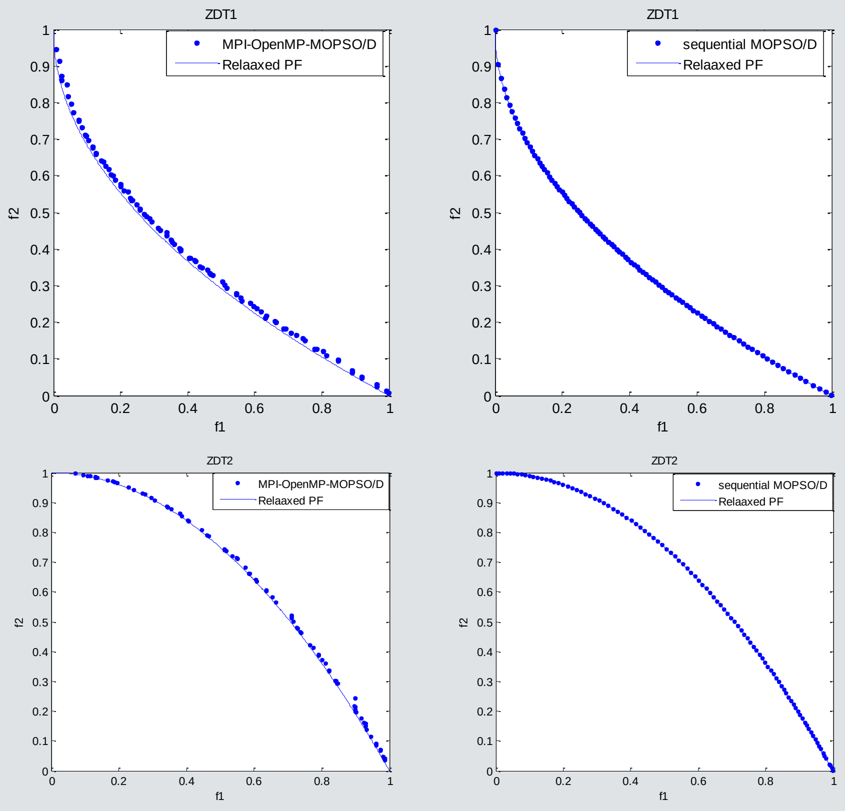
ZDT1	1404	715	1.96	3.27
ZDT2	1349	662	2.04	3.40
ZDT3	1391	703	1.98	3.30
ZDT4	1449	1266	1.14	3.43
ZDT6	1495	1294	1.16	3.47

Table III shows the mean IGD obtained by MPI-OpenMP-MOPSO/D, serial MOPSO/D and NSGA-II for each test instance. From Table III, the proposed algorithm gains a slightly better solution on ZDT3 and a little worse solutions on the other instances than the serial MOPSO/D. Compared with NSGA-II, MPI-OpenMP-MOPSO/D has great advantage in the quality of solutions.

TABLE III. COMPARISION OF IGD AMONG THESE THREE ALGORITHMS

Test Instance	MPI-OpenMP-MOPSO/D	serial MOPSO/D	NSGA-II
ZDT1	1.32E-2	3.897E-3	5.059E-2
ZDT2	9.86E-3	3.826E-3	7.573E-3
ZDT3	7.76E-3	5.351E-3	3.572E-2
ZDT4	7.16E-3	4.051E-3	2.527E-2
ZDT6	8.10E-3	4.339E-3	1.650E-2

Figure 3 represents the final solutions obtained by MPI-OpenMP-MOPSO/D and serial MOPSO/D for each test instance. From the Figure 3 in terms of the uniformness of solutions, MPI-OpenMP-MOPSO/D is better than the serial MOPSO/D on ZDT3, almost the same as the latter on ZDT1, ZDT2 and ZDT6, and worse than it on ZDT4, which is consistent with the IGD values in Table III.



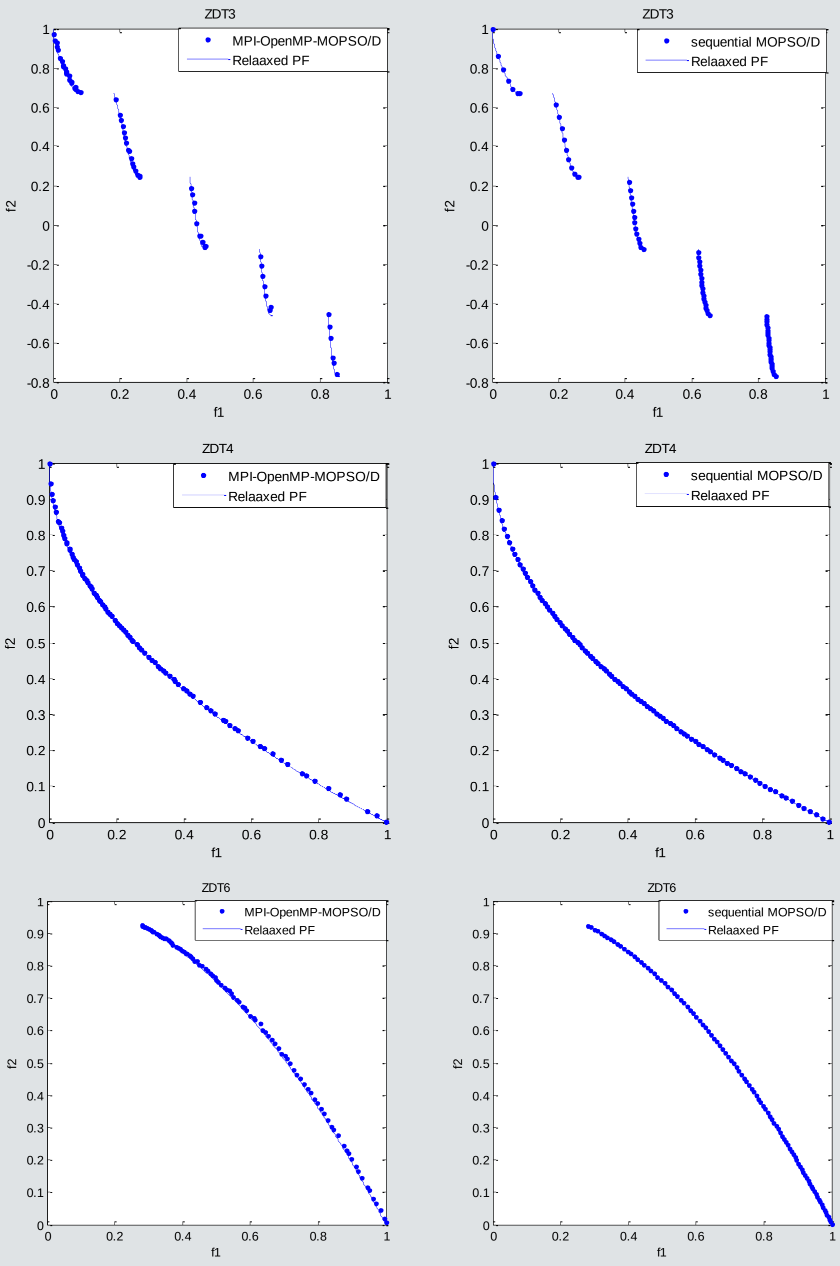


Fig. 3. The final solutions obtained by MPI-OpenMP-MOPSO/D and the serial MOPSO/D for each test instance.

From the above experiment results, we can conclude that MPI-OpenMP-MOPSO/D can obtain slightly worse solutions than the serial MOPSO/D while it only costs much less run-time than the latter. Compared with other MOGA like NSGA-II, the proposed algorithm achieves a good performance on both run-time and the quality of solutions.

V. A Few Improvement to MPI-OpenMP-MOPSO/D

In Section IV, we compare the performance of MPI-OpenMP-MOPSO/D and the serial MOPSO/D, including average run-time and the quality of solutions obtained by these two algorithms. Revisiting Figure 3 and Table III, in terms of solution uniformness, the proposed parallel algorithm performs worse than the serial one on all test instances except ZDT3. So in this section, using a new particle selecting strategy, we improve the uniformness of solutions found by MPI-OpenMP-MOPSO/D without efficiency degradation. On the other hand, we reduce the run-time of the parallel algorithm by running it on a cluster.

A. Deterministically Selecting a Particle to Exchange

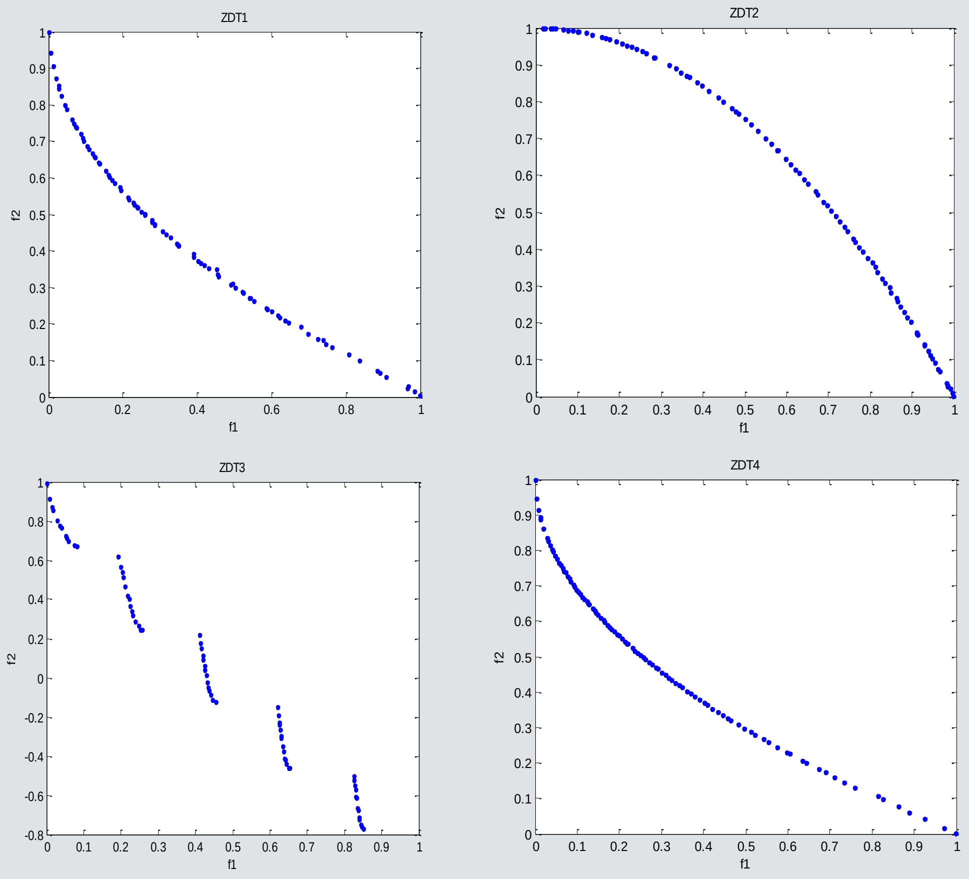
As introduced in Section III-B&C, the left particle and the right one in a subspecies need be send to its left neighbor subspecies and right neighbor, respectively. Meanwhile, this particle can receive two particles from its left and right neighbors, which are used for the evolutions of particles. From this point of view, the exchanged particles have an enormous influence upon the final solutions found by the algorithm.

In the above experiment, we used a naive strategy that one particle randomly selected from the left half of a subspecies and the other one randomly selected from the right half of the subspecies are used as the exchanged particles. While here we deterministically select the first (left) particle and the last (right) particle from a subspecies, then send them to its left neighbor subspecies and right neighbor, respectively.

Table IV shows the mean IGD obtained by the improved MPI-OpenMP-MOPSO/D for each test instance. Compared with the original MPI-OpenMP-MOPSO/D, the improved MPI-OpenMP-MOPSO/D reduces IGD and improves the quality of the final solutions. Fig. 4 represents the approximate Pareto front obtained by the improved MPI-OpenMP-MOPSO/D for each test instance. From Fig. 3 & 4, the uniformness of solutions is improved.

TABLE IV. COMPARISION OF IGD BETWEEN TWO PARALLEL ALGORITHMS WITH RANDOMLY AND DETERMINISTICALLY SELECTING THE EXCHANGED PARTICLES

Instance	the Improved Parallel Algorithm	the Original Parallel Algorithm	the Serial MOPSO/D
ZDT1	1.04E-2	1.32E-02	3.897E-3
ZDT2	5.24E-3	9.86E-03	3.826E-3
ZDT3	5.02E-3	7.76E-03	5.351E-3
ZDT4	6.97E-3	7.16E-03	4.051E-3
ZDT6	5.25E-3	8.10E-03	4.339E-3



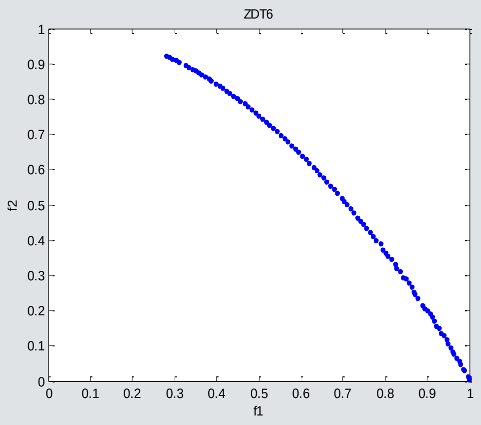


Fig. 4. The final solutions obtained by the improved MPI-OpenMP-MOPSO/D for each test instance.

B. Running on the Cluster

To evaluate the performance of the parallel algorithm without being constrained by the number of processors in a PC, we execute the algorithm on a cluster. The specifications of the cluster are as follows.

- One hundred nodes (only five can be used in the experiment), each is equipped with an Intel Core dual-core 4-thread i3-4150 CPU at 3.5 GHz.
- 4 GB main memory per node.
- Gigabit network with XXXX switches.
- Ubuntu Desktop 12.04 operating system.

Table V shows the average run-time of MPI-OpenMP-MOPSO/D on the cluster (cluster MOPSO/D). Compared with the parallel MOPSO/D executed on a single machine, there is not significant decrease in the run-time for cluster MOPSO/D. Since the increase of communication costs on the cluster offsets the decrease of operation costs, so the total run-time is not greatly reduced. However, cluster can be highly necessary to solve the extremely complex multiobjective optimization problems.

TABLE V. AVERAGE RUN-TIME OF CLUSTER MOPSO/D AND SPEEDUP

Instance	Average run-time (in millisecond) used by			Speedup
	cluster MOPSO/D	stand-alone MOPSO/D	serial MOPSO/D	
ZDT1	626	715	1404	2.24
ZDT2	656	662	1349	2.06
ZDT3	677	703	1391	2.05
ZDT4	1075	1266	1449	1.35
ZDT6	1038	1294	1495	1.44

VI. Conclusion

Multiobjective Particle Swarm Optimization Based on Decomposition (MOPSO/D), an implementation of MOEA/D using PSO, is an effective algorithm for solving MOPs. Based on decomposition, it finds the approximation of Pareto front by solving a number of scalar objective optimization subproblems. In this paper, based on MPI and OpenMP parallel programming platforms, we proposed a parallel algorithm to the serial MOPSO/D, called MPI-OpenMP-MOPSO/D. The algorithm combined distributed-memory and shared-memory programming into an algorithm using MPI and OpenMP, which can fully use the processing power of multicore CPUs and even a cluster.

The proposed algorithm first decomposes the MOP into many scalar optimization subproblems, each of which can be solved by a single particle. Then the particles are decomposed into several subspecies based on the distances of their weight vectors. The evolution of a subspecies is executed by a single process and they communicate with each other by using MPI communication functions. In the process, threads are used to improve the evolution efficiency. In terms of the quality of solutions, the experiment results show MPI-

OpenMP-MOPSO/D can obtain the final solutions which are slightly worse than those of MOPSO/D but better than those of NSGA-II. As for the run-time, the algorithm can achieve speedups of 2x on the single PC equipped with a dual-core four-thread CPU.

Since the uniformness of solutions is not good, the deterministic particle selecting strategy is used in the improved parallel algorithm. The results have shown the strategy highly improved the uniformness of solutions. We have also executed the improved parallel algorithm on a cluster to reduce its run-time. And the result showed that running the algorithm on a cluster can boost its efficiency a bit for these simple test instances, since the communication costs is higher than its operation costs. However, it also suggested that a cluster is more suitable for the complex MOPs. Our future attempts will be in solving the complex MOPs on clusters based MPI-OpenMP-MOPSO/D and accelerating the parallel PSOs via GPU.